Increasing the Accuracy of the Results from the Reciprocal and Reciprocal Square Root Instructions using the Newton-Raphson Method

Version 2.1

01/99

Order Number: 243637-002

Table of Contents

Revision History

Revision	Revision History	Date
1.0	Original publication of document	3/98
2.0	Document revised to reflect actual performance on Si	7/98

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document.

1. *Numerical Recipes in C,* by William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling (Copyright Cambridge University Press 1988)

1 Introduction

This application note discusses the implementation of the Newton-Raphson Method to increase the accuracy of the results for the reciprocal (`rcpps`) and the reciprocal square root (`rsqrtps`) instructions. The `rcpps` and `rsqrtps` instructions return a result, which is accurate in the 12 most significant bits of the mantissa. These two instructions have a 3-cycle latency opposed to 26 cycles required to use the divide instruction.

In some algorithms, it may be desirable to have full accuracy while realizing the performance benefit of using the approximation instructions. The method shown in this application note yields near full accuracy, and provides a sizable performance gain compared to using the divide or square root functions. One iteration of the Newton Raphson method is sufficient to produce a result which is accurate to 23 of 24 bits for single precision numbers (24 bits includes the implied "1" before the binary point

Code for the implementations of the reciprocal (rcpps.cpp) and reciprocal square root (rsqrtps.cpp) are located on the SDK CD-ROM in the \samples\NRReciprocal directory. This code illustrates the Newton Raphson Method implemented in a loop to process an array of floating point values by performing the inverse (`rccps`) or inverse square root (`rsqrtps`). These instructions are not typically used in a loop designed solely to do an inverse. However, the optimized loop provides an example of code scheduling that can be applied to other algorithms that need the additional accuracy provided by the Newton-Raphson Method.

This document also provides a performance comparison between the implementations in this paper and other implementations that are available for Intel® processors.

2 Newton-Raphson Method

The Newton-Raphson formula for finding the root of an equation is defined as:

$$x_{i+1} = x_i - f(x_i) / f'(x_i) \quad (1)$$

where x_i is the estimated root, $f(x_i)$ is the function evaluated at the root estimate and $f'(x_i)$ is the first derivative of the function evaluated at the root estimate. The Newton-Raphson method is the preferred method for finding the root of functions for which the derivative can be easily evaluated and for which the derivative is continuous and non-zero in the neighborhood of the root (see Section 9.4 of *Numerical Recipes in C* by William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, Copyright Cambridge University Press 1988). The Newton-Raphson method approximately doubles the number of significant digits for each iteration if the initial guess is close to the root.

Formula Derivation for 1/x

The goal is to find an equation which, when solved for the root, yields the inverse of a number. Choosing f(x) = 1/x - a and solving for the root yields:

```
1/x - a = 0
1/x = a
x = 1/a
```

So, f(x) = 1/x - a. Taking the first derivative of f(x) yields $f'(x) = -(1/x^2)$. We can now substitute f(x) and f'(x) into the Newton-Raphson formula.

$$x_{i+1} = x_i - \{f(x_i) / f'(x_i)\}$$
$$x_{i+1} = x_i - \{((1/x_i)- a)/[-(1/x_i^2)]\} = x_i - \{((1/x_i)- a)*[-(x_i^2)]\}$$

Simplifying this equation yields:

$$x_1 = x_0 - (a * x_0^2 - x_0)$$

Let $x_0 = rcpps(a)$, which is our initial guess, in the Newton-Raphson formula and substitute.

$$X_1 = rcpps(a) - (a * rcpps(a) * rcpps(a) - rcpps(a)$$
$$x_1 = rcpps(a) - (a * rcpps(a) * rcpps(a) - rcpps(a))$$
$$x_1 = 2 * rcpps(a) - a * rcpps(a) * rcpps(a) \quad (2)$$

The code presented in this paper implements Equation (2). An alternative equation that could be implemented is shown as Equation (3), below.

$$x_1 = rcpps(a) * [2 - a * rcpps(a)] \quad (3)$$

The total latency of 16 cycles (assuming the data to inverse is in a register) is the same for either implementation. Equation (2) has two Port 0 instructions and three Port 1 instructions. Equation (3) has two Port 0 instructions, two Port 1 instructions, and one Port 2 instruction. Choosing which implementation to use should be based on the port usage of the code that surrounds the reciprocal code.

Formula Derivation for 1/sqrt(x)

The goal is to find an equation which, when solved for the root, yields the square root of the inverse of a number. Choosing $f(x) = 1/x^2 - a$ and solving for the root yields:

$$1/x^2 - a = 0$$
$$x^2 = 1/a$$

So, $x = sqrt(1/a)$, where sqrt( ) indicates the sqrt function. Therefore, $f(x) = 1/x^2 - a$. Taking the first derivative of $f(x)$ yields $f'(x) = -2x^{-3}$. Now $f(x)$ and $f'(x)$ can be substituted into the Newton-Raphson formula. This yields:

$$x_{i+1} = x_i - f(x_i) / f'(x_i)$$
$$x_{i+1} = x_i - (1/x_i^2 - a)/-2x^{-3} = x_i - (1/x_i^2 - a)* -0.5x_i^3 = x_i -(x_i - a x_i^3)* -0.5$$
$$x_1 = x_0 - (0.5 * a * x_0^3 - 0.5 * x_0)$$

Now, let $x_0 = rsqrtps(a)$, and substitute this into the previous equation, yielding:

$$x_1 = rsqrtps(a) - (0.5 * a * rsqrtps(a)^3 - 0.5 * rsqrtps(a))$$

Combining the terms results in:

$$x_1 = 1.5 * rsqrtps(a) - 0.5 * a * rsqrtps(a)^3$$

Factoring out a 0.5 and a rsqrtps results in:

$$x_1 = 0.5 * rsqrtps(a) * [3.0 - (a * rsqrtps(a))* rsqrtps(a)] \quad (4)$$

Note that the evaluation order is important in Equation (4). The term, `a * rsqrtps(a)`, should be evaluated first instead of the term, `rsqrtps(a) * rsqrtps(a)`, to preserve accuracy over the full range of single precision numbers. The code presented in this paper implements Equation (4).

2.1 Error Derivation for the Newton-Raphson Method

This section presents a numerical proof of the accuracy obtained for each of the above equations (numbers 2 and 4). The absolute value of the error for the output of the approximation instructions is defined to be less than or equal to $1.5 * 2^{-12}$ in the mantissa of the result. Let $\varepsilon = 1.5 * 2^{-12}$, then the result of the approximation instructions can be represented as the true result times $(1 \pm \varepsilon)$.

2.1.1 Error Derivation for 1/x

Let 1/a represent the exact mathematical result of the inverse. Then rcpps(a) = 1/a $(1 \pm \varepsilon)$. Substituting this result into Equation (2) gives the following derivation.

$$x_1 = 2 * rcpps(a) - a * rcpps(a) * rcpps(a) \quad (2)$$

$$x_1 = 2 * 1/a(1 \pm \varepsilon) - a * 1/a(1 \pm \varepsilon) * 1/a(1 \pm \varepsilon)$$

$$x_1 = 2 * 1/a(1 \pm \varepsilon) - 1/a(1 \pm \varepsilon)^2 = 1/a\{2*(1 \pm \varepsilon) - (1 \pm \varepsilon)^2\}$$

$$x_1 = 1/a\{2 \pm 2\varepsilon - (1 \pm 2\varepsilon + \varepsilon^2)\} = 1/a\{2 \pm 2\varepsilon - 1 \pm (-2\varepsilon) - \varepsilon^2)\}$$

$$x_1 = 1/a\{(2 - 1) \pm (2\varepsilon - 2\varepsilon) - \varepsilon^2)\}$$

$$x_1 = 1/a(1 - \varepsilon^2) \quad (5)$$

Equation (5) shows that the maximum error is in the range, $-\varepsilon^2 <= MaxError <= 0.\varepsilon^2 = 1.125 * 2^{-23}$; therefore, the result will be accurate to 23 bits (out of the total of 24 bits). This accuracy is obtained for all values of α with base 2 exponents of 125 or less. If the base 2 exponent is 126, then the accuracy of the result is 21 out of 24 bits, and if the base 2 exponent is 127, the accuracy of the result is 19 out of 24 bits. This result is valid if flush-to-zero mode is not enabled. If flush-to-zero mode is enabled, then the reciprocal instruction generates a zero result for base 2 exponents of 126 or 127.

2.1.2 Error Derivation for sqrt(1/x)

Let sqrt(1/a) represent the exact mathematical result of the inverse. Then rsqrtps(a) = sqrt(1/a) $(1 \pm \varepsilon)$. Substituting this result into Equation (4) gives the following derivation.

$$x_1 = 0.5 * rsqrtps(a) * [3.0 - (a * rsqrtps(a))* rsqrtps(a)] \quad (4)$$

$$x_1 = 0.5 * sqrt(1/a)(1 \pm \varepsilon) * [3.0 - (a * (sqrt(1/a)(1 \pm \varepsilon))^2)]$$

$$x_1 = 0.5 * sqrt(1/a)(1 \pm \varepsilon) * [3.0 - (a * (1/a)(1 \pm 2\varepsilon + \varepsilon^2))]$$

$$x_1 = 0.5 * sqrt(1/a)(1 \pm \varepsilon) * [2 - (\pm 2\varepsilon) - \varepsilon^2]$$

$$x_1 = 0.5 * sqrt(1/a) * [2 - (\pm 2\varepsilon) - \varepsilon^2 \pm 2\varepsilon - 2\varepsilon^2 - (\pm\varepsilon^3)]$$

$$x_1 = 0.5 * sqrt(1/a) * [2 - 3\varepsilon^2 - (\pm\varepsilon^3)]$$

$$x_1 = sqrt(1/a) * [1 - 1.5\varepsilon^2 - (\pm 0.5\varepsilon^3)]$$

The ε^3 term in the previous equation is out of range and can be dropped, which results in:

$$x_1 = \text{sqrt}(1/a)(1 - 1.5\varepsilon^2) \quad (6)$$

Equation (6) shows that the maximum error for the mantissa is in the range, $-1.5\varepsilon^2 <= \text{MaxError} <= 0$; therefore, the maximum error is $1.6875 * 2^{-23}$. This result is accurate to 23 bits (out of a total of 24 bits). This accuracy is obtained for all values of α with any valid base 2 exponent as long as the evaluation order is preserved in the code. Denormal numbers will exhibit a larger error. The programmer must write code to check for this condition if it is a concern.

2.1.3 A Faulty Optimization Technique for sqrt(1/x)

At first, it may appear that the Newton-Raphson equation for sqrt(1/x) can be simplified by substituting `rcpps(a)` for `rsqrtps(a)`^{2} in Equation (4) to save a multiply. Doing so would result in the following equation.

$$x_1 = 0.5 * \text{rsqrtps}(a) * [3.0 - (a * (\text{rcpps}(a))] \quad (7)$$

Such an attempt leads to incorrect results. To illustrate this, let:

ε_1 = the error for the rcpps instruction

ε_2 = the error for the rsqrtps instruction

Then, as shown earlier, let:

$\text{rcpps}(a) = 1/a(1 \pm \varepsilon_1)$, and

$\text{rsqrtps}(a) = \text{sqrt}(1/a)(1 \pm \varepsilon_2)$

Finally, by substituting the above two equations into (7), we have the equation for x_1:

$$x_1 = 0.5 * \text{sqrt}(1/a)(1 \pm \varepsilon_2) * [3.0 - (a * (1/a)(1 \pm \varepsilon_1))]$$

$$x_1 = 0.5 * \text{sqrt}(1/a)(1 \pm \varepsilon_2) * [2.0 - (\pm \varepsilon_1)]$$

$$x_1 = 0.5 * \text{sqrt}(1/a)[2.0 - (\pm \varepsilon_1) \pm 2\varepsilon_2 \pm \varepsilon_2\varepsilon_1]$$

$$x_1 = \text{sqrt}(1/a)[1.0 - (0.5 \pm \varepsilon_1) \pm \varepsilon_2 \pm 0.5\varepsilon_2\varepsilon_1]$$

Since ε_1 and ε_2 are independent, then in general, the above terms do not cancel and the result of this equation is approximately the same accuracy as the original `rsqrtps` instruction. Therefore, this optimization attempt does not work.

2.2 Implementing the Newton-Raphson Method

Typically, the algorithms presented in this paper would be used within a function that has a greater purpose than that of simply doing the reciprocal or reciprocal square root. Realizing the optimal performance from the Newton Raphson equations derived in Section 2 requires that other instructions for concurrent scheduling are available in order to hide the latency of the instructions used in the Newton-Raphson method. This application note presents the instructions required to implement the Newton-Raphson method in an unoptimized form. These instructions should be added to the programmer's function and scheduled to produce the best performance. A simple test program is included for both equations that illustrates the advantages of instruction scheduling by unrolling the

loop and performing the Newton-Raphson method on an array of data (see Section 3.3). The unrolled loop performance data is presented to give the programmer an idea of the performance gain possible due to optimal scheduling of instructions.

2.3 Accuracy of Results

The methods presented here result in a minimum of 23 bits of accuracy (out of 24 bits total) in the single precision result. One exception exists for the reciprocal code (although it does not for the reciprocal square root code): when the base 2 exponent is greater than or equal to 126, the accuracy of the reciprocal decreases (refer to Section 2.1.1 for additional information). Denormals result in less accurate results. The programmer must determine if checking for denormals (or any other exceptional condition) is required for any particular implementation.

2.4 Implementation Requirements

The order of multiplies must be observed and duplicated as presented in this application note to preserve accuracy. Evaluation of the multiplies in the wrong order can result in an overflow or underflow condition, or in reduced accuracy of the results. Comments are included in the source code to point out the areas where evaluation order is important.

3 Conclusion

Significant performance improvements are possible by using the reciprocal and reciprocal square root approximation instructions along with the Newton-Raphson method. When full single precision is required for a reciprocal or a reciprocal square root, the algorithm(s) discussed in this paper are a much better choice compared to C code or the `divps` and `sqrtps` SIMD instructions. However, because of the superior performance, the reciprocal approximation instructions should be used by themselves if the accuracy is sufficient.

4 Code Samples

Code samples for the implementations of the reciprocal (rcpps.cpp) and reciprocal square root (rsqrtps.cpp) are located on the Intel SDK CD-ROM in the \samples\NRReciprocal directory.